DESIGN PATTERNS PROJECT REPORT

Sports Article Querying System

Using the Composite, Strategy and Iterator patterns.

**Team Members**

Shravya Kadur 01FB14ECS227

Shilpa S Nair 01FB14ECS221

**Introduction**

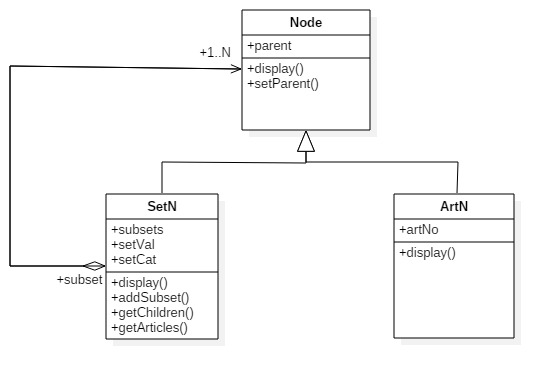
The Sports Article Querying System creates a tree that arranges sports news articles by category and allows it to be queried, the result of which is a subtree with articles related to the query.

The initial dataset was obtained from Kaggle and contains **1408** sports news articles. The dataset was cleaned and processed to obtain more details related to it. The names of sports were identified and stored for each article by looking for the presence of the sport type in the article itself. In case of absence, sport type-specific terms were found and correlated. Using the **Stanford NER** (a Named Entity Recognition) system, names of people, locations and organisations in the articles are recognized, extracted and stored as new columns of attributes for the articles. This could result in null values in certain cases, when the article didn’t specify any one of the above and in the few cases where the NER misclassifies or misses data.

The patterns used for this application are the **Composite** pattern, the **Strategy** pattern and the **Iterator** pattern. The Tree class encapsulates these by holding the root of the tree (implementation of the composite pattern), a query system object (implementation of the strategy pattern that in turn makes use of Iterator class which implements the iterator pattern) and a lookupthat stores all types of names in the tree and maps them to their types (sport, person, location or organisation) to aid in querying.

**Patterns Used**

**1. Composite**

****

The composite design pattern helps us compose objects into tree structures to represent part-whole hierarchies. The clients are able to treat individual components and compositions of components in the same manner. In this application, we make use of a set of articles which we decompose into multiple sets of articles that are related.

Here the tree can have two types of nodes: **Article Nodes** and **Set Nodes**. The leaf nodes of the trees are Article Nodes while the Set Nodes are compositions of the other nodes (can contain other Set Nodes and Article Nodes). The main functionality needed by the client is to view the article or all articles in the set and this is implemented as needed in both types of nodes to make it usable to the client uniformly.

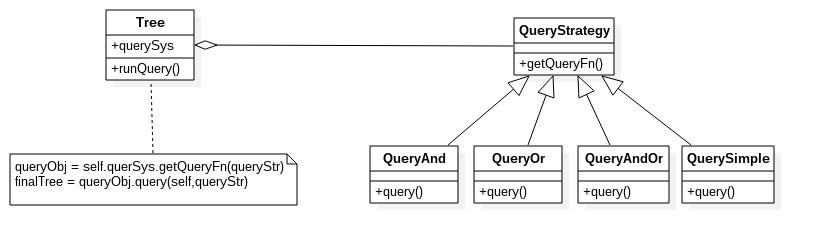
The Article Nodes do not hold the entire article but note the line of the article in the data so that it can access that specific line while displaying. This way, we avoid holding the entire dataset of 1408 in memory while running and this fact is helpful if we wish to expand the data. While making the tree, we create multiple article nodes per article as needed as articles can fall under different sets.

The Set Nodes hold data pertaining to the semantics of the set and functions needed to handle the children of that set, along with a display function that in turn displays all children such that articles are displayed uniquely. Sets are grouped based on a common attribute that can be searched for. The types or categories of these nodes are- sport, location, person and organisation. The root node is also a set node and is the only node of a different category in the initial tree, ‘root’ and after querying the resulting root node is of a category named ‘reroot’. The tree is organised such that sets follow that order according to what attributes the article has. For example, if an article has sport and person only, it is guaranteed that the sport set contains the person set which in turn contains the article.

Classes involved:

* Node
* SetN
* ArtN

**2. Strategy**

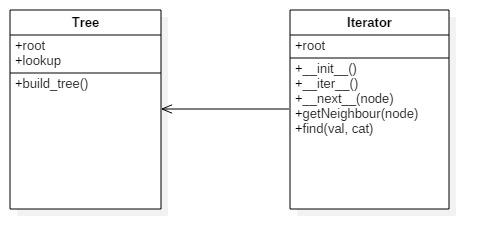
The strategy design pattern defines a family of algorithms, encapsulate each, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. Here we are using the strategy design pattern to have different ways of handling simple queries (querying for different attributes) and different types of complex queries (querying for multiple attributes combined in different ways)

The Tree class provides the context. It contains an object of QueryStrategy and when the function runQuery() is called on a tree object, it uses this object to obtain a querying object that encapsulates the querying technique required for that kind of query. The different kinds of complex queries involve: simple queries combined by ‘or’, ‘and’ or both and simple queries contain only one attribute. Each technique constructs the resultant tree differently and returns a tree (composite pattern) as its return type. The objects returned could be of any of its subclasses, i.e. QueryAnd, QueryOr, QueryAndOr or QuerySimple.

Classes involved:

* Tree
* QueryStrategy
* QueryAnd
* QueryOr
* QueryAndOr
* QuerySimple

**3. Iterator**

* ****

The Iterator pattern is implemented using the Iterator class, which provides a way to access the elements of the Tree(which is an aggregate object) sequentially, without exposing its underlying representation.

The main functionality of this class is to run through the tree and find set nodes with the corresponding category-value pair requested by one of the QueryStrategy classes.

The class accomplishes this by using the find() method, which is based on the **Depth First Search** Algorithm. This is very suited for our tree since we can easily ignore the children of the node that matches our search, and not waste computational time by running through them.

It also consists of other methods that iterate through the tree, such as -

\_\_next\_\_(node) - gets the next unvisited child of the node.

getNeighbour(node) - returns the neighbour, i.e. the next node on the same level as the input node.

Classes involved:

Tree

Iterator

**Results**

In this application, we created a tree that organizes sports articles by categories based on attributes that we could extract. The original tree consisted of various nodes, comes up to a maximum of **five levels** (including the root), as we have four different categories of attributes. The tree can be queried in different ways, using the attributes extracted (alone or combined).